Our Ref.: 723-968

U.S. PATENT APPLICATION

Inventor(s): Robert A. Drebin
Timothy J. Van Hook
Patrick Y. Law
Mark M. Leather
Matthew Komsthoeft

Invention: RECIRCULATING SHADE TREE BLENDER FOR A GRAPHICS SYSTEM

NIXON & VANDERHYE P.C.
ATTORNEYS AT LAW
1100 NORTH GLEBE ROAD
8TH FLOOR
ARLINGTON, VIRGINIA 22201-4714
(703) 816-4000
Facsimile (703) 816-4100

SPECIFICATION

blending operations, are supported for applying multiple textures to a single object in a single rendering pass.

0882-1-9622-960

Recirculating Shade Tree Blender For A Graphics System

Ins. a1

~~This application claims the benefit of U.S. Provisional Application serial No. 60/226,888, filed August 23, 2000, the entire content of which is hereby incorporated by reference in this application.~~ ~~This application is related to concurrently-filed application Serial No. _____ of Law et al entitled "Method And Apparatus For Providing Logical Combination Of N Alpha Operations Within A Graphics System" (attorney ref. 723-973).~~

Field of the Invention

The present invention relates to computer graphics, and more particularly to interactive graphics systems such as home video game platforms. Still more particularly this invention relates to recirculating shader hardware for implementing shade trees for multitexturing and other effects.

Background And Summary Of The Invention

Many of us have seen films containing remarkably realistic dinosaurs, aliens, animated toys and other fanciful creatures. Such animations are made possible by computer graphics. Using such techniques, a computer graphics artist can specify how each object should look and how it should change in appearance over time, and a computer then models the objects and displays them on a display such as your television or a computer screen. The computer takes care of performing the many tasks required to make sure that each part of the displayed image is colored and shaped just right based on the position and orientation of each object in a scene, the direction in which light seems to strike each object, the surface texture of each object, and other factors.

Because computer graphics generation is complex, computer-generated three-dimensional graphics just a few years ago were mostly limited to expensive specialized flight simulators, high-end graphics workstations and supercomputers. The public saw some of the images generated by these computer systems in movies and expensive television advertisements, but most of us couldn't actually interact with the computers doing the graphics generation. All this has changed with the availability of relatively inexpensive 3D graphics platforms such as, for example, the Nintendo 64® and various 3D graphics cards now available for personal computers. It is now possible to interact with exciting 3D animations and simulations on relatively inexpensive computer graphics systems in your home or office.

A problem graphics system designers confronted in the past was how to efficiently implement shaders in a graphics system. Generally, shading is the process performing lighting computations and determining pixel colors/opacities from them. Generally, there are three main types of shading in common use: flat, Gouraud, and Phong. These correspond to computing the light per polygon, per vertex and per pixel. A wide variety of shading models have been created. There is no one shading model that pleases all users and is suitable for all applications. Therefore, several design approaches have been suggested to provide flexibility in terms of programmer selection and specification of shading models.

In the paper by R.L. Cook called "Shade Trees" (*SIGGRAPH 84*, pages 223-231, the author described a special purpose language in which a shader is built as a tree expression called a shade tree. Generally speaking, a shade tree is a tree of nodes each of which takes parameters from its children and produces parameters for its parent. For example, the parameters may be the terms of the illumination equation (e.g., specular coefficient or surface Normal). Other parameters might

comprise atmospheric effects (e.g., haze) or projections. The RenderMan Interface uses shade trees to provide user-defined and system-defined shaders for a variety of purposes.

While shade trees have been used extensively in non-real-time rendering graphics systems, problems arise when trying to accommodate the flexibility that shade trees provide within the context of real-time rendering. It would be highly desirable to be able to provide the flexibility of shade trees within low cost real-time rendering systems such as, for example, home video game platforms and personal computer graphics cards.

Another problem confronting graphics systems designers has been how to efficiently provide a feature called single-pass multitexturing. Basically, texturing is a technique for efficiently modeling the properties of a surface. For example, instead of modeling the geometry of each individual brick and mortar line within a brick wall, it is possible to electronically "glue" an image of a brick wall onto a surface. Such texturing capabilities can be used to significantly increase image complexity without a corresponding increase in modeling and processing costs.

The extension to texturing known as multitexturing allows two or more textures to be applied to the same surface. For example, suppose you want to create an image of the earth as it might be seen from outer space. You could model the earth as a sphere and apply two different textures to it. The first texture could be an image of the continents and oceans. The second texture could be an image of cloud cover. By moving the cloud cover texture image relative to the continent/ocean texture image, you could create a very realistic dynamic texture-mapped image.

Some graphics accelerators support multitexturing in which two or more textures are accessed during the same rendering pass. See, for example,

Microsoft's Direct X 6.0 SBK (1998); Segal et al., "The Open GL Graphics System: A Specification" (Version 1.2.1) (March 1998) (www.OpenGL.org). Certain PC graphics accelerator cards also provide single pass multitexturing. However, further improvements are possible.

The present invention provides a generalized shade tree blender that can be used for multitexturing as well as a number of other flexible blending effects. In accordance with one aspect provided by this invention, recirculating shader hardware within a graphics pipeline can be controlled to provide a number of independently controllable blending stages. A shader hardware includes intermediate storage for results of previous blending operations. The shader hardware can select different inputs and perform different operations for each blending stage. Thus, relatively low cost and compact shader hardware can be used to implement arbitrarily complex shade trees.

In accordance with another aspect provided by this invention, the results of a first texture mapping operation is provided to a reconfigurable shader. The shader performs a blending operation in response to the first texture mapping operation. The shader is then reconfigured, and is connected to receive the results of a further texturing operation. The reconfigured shader combines its previous results with the results of the further texturing operation to provide a blended output.

In accordance with a further aspect provided by this invention, a shader can be recirculated any desired number of times to implement an arbitrarily complex shading model. Each recirculation or "stage" can be programmed to have any one of a number of desired blending operations and to blend from selected ones of a variety of color, opacity or depth sources. The number of recirculations may be limited in a particular implementation in view of real-time rendering timing

constraints, but a reasonable number of recirculation stages (e.g., fifteen) can provide great flexibility in implementing a variety of complex shading models.

In accordance with another aspect provided by this invention, a recirculating shade tree pixel blender is implemented in hardware to minimize processing time per stage. In more detail, a preferred embodiment of this invention provides a relatively low chip-footprint, versatile texture-environment processing subsystem including a hardware accelerated programmable texture shader/pixel blender that circulates computed color, opacity and other data over multiple cycles/stages. The texture environment subsystem can combine per-vertex lighting, textures, rasterized colors, opacities, and depths to form pixel parameters for display. Blending operations for color (e.g., RGB) and alpha components may be independently processed within the texture environment subsystem by a blending unit comprising a set of color/alpha combiner (shader) hardware that is reused over multiple processing stages to implement multitexturing and other effects. Selectable current-color/opacity input/output registers may be shared among all stages to store intermediate results. The shader hardware can be reconfigured for each stage to provide a chain of specifiable blending/shading operations supporting single rendering pass multitexturing and other effects.

Brief Description Of The Drawings

These and other features and advantages provided by the invention will be better and more completely understood by referring to the following detailed description of presently preferred embodiments in conjunction with the drawings, of which:

Figure 1 is an overall view of an example interactive computer graphics system;

Figure 2 is a block diagram of the Figure 1 example computer graphics system;

Figure 3 is a block diagram of the example graphics and audio processor shown in Figure 2;

Figure 4 is a block diagram of the example 3D graphics processor shown in Figure 3;

Figure 5 is an example logical flow diagram of the Figure 4 graphics and audio processor;

Figure 6 shows an example reusable recirculating shader;

Figure 7 shows an example shading pipeline implemented using the recirculating shader;

Figure 8 shows an example recirculating shader block diagram;

Figure 9 shows an example recirculating shader input multiplexer;

Figure 10 shows an example recirculating shader operation block diagram;

Figure 11 shows an example recirculating shader implementation;

Figures 12A and 12B illustrate an example color swap feature;

Figure 13 shows an example texture environment unit implementation;

Figure 14 shows an example fog calculation unit;

Figure 15 shows an example of how the recirculating shader can be used for multitexturing;

Figure 16 shows an example multitexturing process using the recirculating shader;

0022-7-0022-0022-US

Figure 17 shows an example multi-texture pipeline using the recirculating shader;

Figure 18 shows an example multi-texture pipeline control;

Figure 19 shows example texture environment unit control registers; and

Figures 20A and 20B show example alternative compatible implementations.

Detailed Description Of Example Embodiments Of The Invention

Figure 1 shows an example interactive 3D computer graphics system 50. System 50 can be used to play interactive 3D video games with interesting stereo sound. It can also be used for a variety of other applications.

In this example, system 50 is capable of processing, interactively in real-time, a digital representation or model of a three-dimensional world. System 50 can display some or all of the world from any arbitrary viewpoint. For example, system 50 can interactively change the viewpoint in response to real-time inputs from handheld controllers 52a, 52b or other input devices. This allows the game player to see the world through the eyes of someone within or outside of the world. System 50 can be used for applications that do not require real-time 3D interactive display (e.g., 2D display generation and/or non-interactive display), but the capability of displaying quality 3D images very quickly can be used to create very realistic and exciting game play or other graphical interactions.

To play a video game or other application using system 50, the user first connects a main unit 54 to his or her color television set 56 or other display device by connecting a cable 58 between the two. Main unit 54 produces both video signals and audio signals for controlling color television set 56. The video signals

are what controls the images displayed on the television screen 59, and the audio signals are played back as sound through television stereo loudspeakers 61L, 61R.

The user also needs to connect main unit 54 to a power source. This power source may be a conventional AC adapter (not shown) that plugs into a standard home electrical wall socket and converts the house current into a lower DC voltage signal suitable for powering the main unit 54. Batteries could be used in other implementations.

The user may use hand controllers 52a, 52b to control main unit 54. Controls 60 can be used, for example, to specify the direction (up or down, left or right, closer or further away) that a character displayed on television 56 should move within a 3D world. Controls 60 also provide input for other applications (e.g., menu selection, pointer/cursor control, etc.). Controllers 52 can take a variety of forms. In this example, controllers 52 shown each include controls 60 such as joysticks, push buttons and/or directional switches. Controllers 52 may be connected to main unit 54 by cables or wirelessly via electromagnetic (e.g., radio or infrared) waves.

To play an application such as a game, the user selects an appropriate storage medium 62 storing the video game or other application he or she wants to play, and inserts that storage medium into a slot 64 in main unit 54. Storage medium 62 may, for example, be a specially encoded and/or encrypted optical and/or magnetic disk. The user may operate a power switch 66 to turn on main unit 54 and cause the main unit to begin running the video game or other application based on the software stored in the storage medium 62. The user may operate controllers 52 to provide inputs to main unit 54. For example, operating a control 60 may cause the game or other application to start. Moving other controls 60 can cause animated characters to move in different directions or change the

user's point of view in a 3D world. Depending upon the particular software stored within the storage medium 62, the various controls 60 on the controller 52 can perform different functions at different times.

Example Electronics of Overall System

Figure 2 shows a block diagram of example components of system 50. The primary components include:

- a main processor (CPU) 110,
- a main memory 112, and
- a graphics and audio processor 114.

In this example, main processor 110 (e.g., an enhanced IBM Power PC 750) receives inputs from handheld controllers 108 (and/or other input devices) via graphics and audio processor 114. Main processor 110 interactively responds to user inputs, and executes a video game or other program supplied, for example, by external storage media 62 via a mass storage access device 106 such as an optical disk drive. As one example, in the context of video game play, main processor 110 can perform collision detection and animation processing in addition to a variety of interactive and control functions.

In this example, main processor 110 generates 3D graphics and audio commands and sends them to graphics and audio processor 114. The graphics and audio processor 114 processes these commands to generate interesting visual images on display 59 and interesting stereo sound on stereo loudspeakers 61R, 61L or other suitable sound-generating devices.

Example system 50 includes a video encoder 120 that receives image signals from graphics and audio processor 114 and converts the image signals into analog and/or digital video signals suitable for display on a standard display device such

as a computer monitor or home color television set 56. System 50 also includes an audio codec (compressor/decompressor) 122 that compresses and decompresses digitized audio signals and may also convert between digital and analog audio signaling formats as needed. Audio codec 122 can receive audio inputs via a buffer 124 and provide them to graphics and audio processor 114 for processing (e.g., mixing with other audio signals the processor generates and/or receives via a streaming audio output of mass storage access device 106). Graphics and audio processor 114 in this example can store audio related information in an audio memory 126 that is available for audio tasks. Graphics and audio processor 114 provides the resulting audio output signals to audio codec 122 for decompression and conversion to analog signals (e.g., via buffer amplifiers 128L, 128R) so they can be reproduced by loudspeakers 61L, 61R.

Graphics and audio processor 114 has the ability to communicate with various additional devices that may be present within system 50. For example, a parallel digital bus 130 may be used to communicate with mass storage access device 106 and/or other components. A serial peripheral bus 132 may communicate with a variety of peripheral or other devices including, for example:

- a programmable read-only memory and/or real-time clock 134,
- a modem 136 or other networking interface (which may in turn connect system 50 to a telecommunications network 138 such as the Internet or other digital network from/to which program instructions and/or data can be downloaded or uploaded), and
- flash memory 140.

A further external serial bus 142 may be used to communicate with additional expansion memory 144 (e.g., a memory card) or other devices. Connectors may be used to connect various devices to busses 130, 132, 142.

Example Graphics And Audio Processor

Figure 3 is a block diagram of an example graphics and audio processor 114. Graphics and audio processor 114 in one example may be a single-chip ASIC (application specific integrated circuit). In this example, graphics and audio processor 114 includes:

- a processor interface 150,
- a memory interface/controller 152,
- a 3D graphics processor 154,
- an audio digital signal processor (DSP) 156,
- an audio memory interface 158,
- an audio interface and mixer 160,
- a peripheral controller 162, and
- a display controller 164.

3D graphics processor 154 performs graphics processing tasks. Audio digital signal processor 156 performs audio processing tasks. Display controller 164 accesses image information from main memory 112 and provides it to video encoder 120 for display on display device 56. Audio interface and mixer 160 interfaces with audio codec 122, and can also mix audio from different sources (e.g., streaming audio from mass storage access device 106, the output of audio DSP 156, and external audio input received via audio codec 122). Processor interface 150 provides a data and control interface between main processor 110 and graphics and audio processor 114.

Memory interface 152 provides a data and control interface between graphics and audio processor 114 and memory 112. In this example, main processor 110 accesses main memory 112 via processor interface 150 and memory interface 152 that are part of graphics and audio processor 114. Peripheral

controller 162 provides a data and control interface between graphics and audio processor 114 and the various peripherals mentioned above. Audio memory interface 158 provides an interface with audio memory 126.

Example Graphics Pipeline

Figure 4 shows a more detailed view of an example 3D graphics processor 154. 3D graphics processor 154 includes, among other things, a command processor 200 and a 3D graphics pipeline 180. Main processor 110 communicates streams of data (e.g., graphics command streams and display lists) to command processor 200. Main processor 110 has a two-level cache 115 to minimize memory latency, and also has a write-gathering buffer 111 for uncached data streams targeted for the graphics and audio processor 114. The write-gathering buffer 111 collects partial cache lines into full cache lines and sends the data out to the graphics and audio processor 114 one cache line at a time for maximum bus usage.

Command processor 200 receives display commands from main processor 110 and parses them -- obtaining any additional data necessary to process them from shared memory 112. The command processor 200 provides a stream of vertex commands to graphics pipeline 180 for 2D and/or 3D processing and rendering. Graphics pipeline 180 generates images based on these commands. The resulting image information may be transferred to main memory 112 for access by display controller/video interface unit 164 -- which displays the frame buffer output of pipeline 180 on display 56.

Figure 5 is a logical flow diagram of graphics processor 154. Main processor 110 may store graphics command streams 210, display lists 212 and vertex arrays 214 in main memory 112, and pass pointers to command processor

200 via bus interface 150. The main processor 110 stores graphics commands in one or more graphics first-in-first-out (FIFO) buffers 210 it allocates in main memory 110. The command processor 200 fetches:

- command streams from main memory 112 via an on-chip FIFO memory buffer 216 that receives and buffers the graphics commands for synchronization/flow control and load balancing,
- display lists 212 from main memory 112 via an on-chip call FIFO memory buffer 218, and
- vertex attributes from the command stream and/or from vertex arrays 214 in main memory 112 via a vertex cache 220.

Command processor 200 performs command processing operations 200a that convert attribute types to floating point format, and pass the resulting complete vertex polygon data to graphics pipeline 180 for rendering/rasterization. A programmable memory arbitration circuitry 130 (see Figure 4) arbitrates access to shared main memory 112 between graphics pipeline 180, command processor 200 and display controller/video interface unit 164.

Figure 4 shows that graphics pipeline 180 may include:

- a transform unit 300,
- a setup/rasterizer 400,
- a texture unit 500,
- a texture environment unit 600, and
- a pixel engine 700.

Transform unit 300 performs a variety of 2D and 3D transform and other operations 300a (see Figure 5). Transform unit 300 may include one or more matrix memories 300b for storing matrices used in transformation processing 300a.

Transform unit 300 transforms incoming geometry per vertex from object space to screen space; and transforms incoming texture coordinates and computes projective texture coordinates (300c). Transform unit 300 may also perform polygon clipping/culling 300d. Lighting processing 300e also performed by transform unit 300b provides per vertex lighting computations for up to eight independent lights in one example embodiment. Transform unit 300 can also perform texture coordinate generation (300c) for embossed type bump mapping effects, as well as polygon clipping/culling operations (300d).

Setup/rasterizer 400 includes a setup unit which receives vertex data from transform unit 300 and sends triangle setup information to one or more rasterizer units (400b) performing edge rasterization, texture coordinate rasterization and color rasterization.

Texture unit 500 (which may include an on-chip texture memory (TMEM) 502) performs various tasks related to texturing including for example:

- retrieving textures 504 from main memory 112,
- texture processing (500a) including, for example, multi-texture handling, post-cache texture decompression, texture filtering, embossing, shadows and lighting through the use of projective textures, and BLIT with alpha transparency and depth,
- bump map processing for computing texture coordinate displacements for bump mapping, pseudo texture and texture tiling effects (500b), and
- indirect texture processing (500c).

~~For details concerning the operation of blocks 500a,500b and 500c, a more~~ detailed description ~~of the~~ example graphics pipeline circuitry and procedures for ~~performing regular and indirect~~ texture look-up ~~operations is disclosed in~~

commonly assigned co-pending patent application, Ser. No. _____, entitled "Method And Apparatus For Direct And Indirect Texture Processing In A Graphics System" (attorney docket no. 723-961) and its corresponding provisional application, serial no. 60/226,891, filed August 23, 2000, both of which are incorporated herein by this reference.

In a preferred example arrangement of the graphics pipeline, texture unit 500 is implemented using a "recirculating" texturing pipeline arrangement that uses a single texture address coordinate/data processing unit that interleaves the processing of logical direct and indirect texture coordinate data and provides a texture lookup data feedback path (500d) for recirculating retrieved indirect texture lookup data from a single texture retrieval unit back to the texture address coordinate/data processing unit.

Texture unit 500 outputs filtered texture values to the texture environment unit 600 for texture environment processing (600a). Texture environment unit 600 blends polygon and texture color/alpha/depth, and can also perform texture fog processing (600b) to achieve inverse range based fog effects. Texture environment unit 600 can provide multiple stages to perform a variety of other interesting environment-related functions based for example on color/alpha modulation, embossing, detail texturing, texture swapping, clamping, and depth blending.

As shown in Figure 5, texture environment unit 600a in the example embodiment includes a recirculating shader 602. Recirculating shader 602 in this example comprises a hardware-based general purpose blender that can blend between a number of selected inputs and can retain blended results for further blending in a subsequent blending operation. Recirculating shader 602 in this example comprises reusable blending logic that can implement a number of different blending operations. In the example embodiment, recirculating shader

602 can retain a number of different distinct previous blending results and can blend newly provided values with any of these previously blended results. This allows recirculating shader 602 to implement any arbitrary shade tree through successive recirculated stages. See, for example, Cook, Shade Trees, *SIGGRAPH Proceedings*, pages 223-231 (July 1984). The complexity of the shade tree that recirculating shader 602 can implement is limited by the total number of times recirculating shader 602 can recirculate within a given rendering pass. In the example embodiment, recirculating shader 602 can provide up to fifteen recirculated stages in the example embodiment but different implementations could provide different numbers of recirculating stages.

Once texture environment unit 600 generates a blended color/alpha/z output, pixel engine 700 performs depth (z) compare (700a) and frame buffer pixel blending (700b). In this example, pixel engine 700 stores data into an embedded (on-chip) frame buffer memory 702. Graphics pipeline 180 may include one or more embedded DRAM memories 702 to store frame buffer and/or texture information locally. Z compares 700a' can also be performed at an earlier stage in the graphics pipeline 180 depending on the rendering mode currently in effect (e.g., z compares can be performed earlier if alpha blending is not required). The pixel engine 700 includes a copy operation 700c that periodically writes on-chip frame buffer 702 to main memory 112 for access by display/video interface unit 164. This copy operation 700c can also be used to copy embedded frame buffer 702 contents to textures in the main memory 112 for dynamic texture synthesis effects. Anti-aliasing and other filtering can be performed during the copy-out operation. The frame buffer output of graphics pipeline 180 (which is ultimately stored in main memory 112) is read each frame by display/video interface unit 164.

Display controller/video interface 164 provides digital RGB pixel values for display on display 102.

Example Recirculating Shader

Figure 6 shows a high-level block diagram of an example recirculating shader implementation, and Figure 7 shows a logical diagram of an example shader pipeline that can be implemented using the Figure 6 recirculating shader. As shown in Figure 6, recirculating shader 602 receives a number of different inputs and generates an output that can be fed back to its input.

In more detail, recirculating shader 602 in the example embodiment can select between any of the following types of inputs:

- rasterized color/alpha,
- texture color/alpha,
- computed color/alpha,
- other.

In the example embodiment, the rasterized color/alpha may be provided by rasterizer 400. For example, the rasterized color/alpha can be Gouraud shaded pixels determined by rasterizer 400 based on lighting calculations performed by transform unit 300 on a per-vertex basis. Texture color/alpha may be the result of a direct or indirect texture mapping operation performed by texture unit 500. Recirculating shader 602 blends any/all of these values and/or constants based on a programmable blending operation to provide a computed color/alpha value at its output. This computed color/alpha value can be fed back as an input to recirculating shader 602 for use in a subsequent blending operation.

Each independently controlled recirculation of recirculating shader 602 may be referred to as a "stage." In the example embodiment, recirculating shader 602 is

implemented by high-speed hardware logic in an application-specific integrated circuit (ASIC). Recirculation of a high-speed hardware logic functionality to provide a number of independently-controlled logical stages provides flexibility by allowing any desired number of shading/blending stages while reducing hardware complexity and required integrated circuit real estate. Each recirculating shader 602 stage in the example embodiment computes independent blending operations for color (RGB) and alpha (transparency). The blending computation in the example embodiment can be programmed by the application running on main processor 110 from a number of different operations including, for example:

- modulate,
- modulate 2x,
- modulate 4x,
- add,
- add sign,
- add sign 2x,
- subtract,
- add smooth,
- blend diffuse alpha,
- blend texture alpha,
- blend factor alpha,
- blend current alpha,
- blend texture alpha pre-multiplied,
- modulate alpha and add color,
- modulate color and add alpha,
- modulate inverse alpha and add color,
- modulate inverse color and add alpha,

- specular color and texture (multiple stages),
- embossing (multiple stages),
- detail texture (multiple stages),
- other operations.

Figure 7, shows an example recirculating shading pipeline that can be provided by recirculating shader 602. This logical pipeline provides multiple sequential blending stages each of which can blend a computed color/alpha with a texture color/alpha and/or a rasterized color/alpha. In the example embodiment, up to sixteen different stages of blending operations can be provided with each stage receiving interpolated texture and raster color/alpha. Programming of rasterizer 400 (see Figure 5) specifies which one of eight texture coordinate sets and which one of eight physical textures 504 to use for texture interpolation provided by recirculating shader 602. Rasterizer 400 may also control which one of two rasterized colors is used for a given recirculating shader 602 stage.

Example Recirculating Shader Implementation

Figure 8 shows a block diagram of an example implementation of recirculating shader 602 suitable for use in texture environment block 600. In the example shown, recirculating shader 602 includes a number of input multiplexers 656 for selecting among various different inputs (e.g., texture color/alpha, rasterized color/alpha, and results of prior blending operations) to be operated on by a blending operator 658. The results of blending operator 658 can be stored in any one of four intermediate registers 660 in the example embodiment. Any one of these registers 660 may then be selected by input multiplexers 656 for a further blending operation in a subsequent blending stage.

In the example embodiment, recirculating shader 8 provides a color component operation and a separate alpha component operation for each blending stage. For a given stage, the operation performed by block 658 can be different for the color component and the alpha component. The color and alpha component operations can be independently controlled by the application running on main microprocessor 110. This ability to provide independent color and alpha operation controls for each recirculating stage allows recirculating shader 602 to perform arbitrarily complex alpha (transparency) trees of operations at no additional cost in terms of processing speed. The alpha produced by the last stage of recirculating shader 602 is input to alpha compare block 662. The results of this alpha compare operation 662 can be used to, for example, conditionally mask color and/or z writes to embedded frame buffer 702.

In the example embodiment, the final blended output of the last blending stage is stored in register 660(4) for color output and/or alpha thresholding 662. A z-texturing path is also provided for z-texturing. See copending commonly assigned U.S. Patent Application Serial No. _____,___of Leather et al. entitled "Z-Texturing" (attorney ref. 723-965) and its corresponding Provisional Application No. 60/226,913, filed August 23, 2000, both of which are incorporated herein by reference.

Example Input Multiplexer Configuration

Figure 9 shows an example recirculating shader input multiplexer configuration 656. As shown in Figure 9, each one of four multiplexers 656a ... 656d can select one of a number of different color inputs including:

- the contents of register 660(1),
- the contents of register 660(2),

- the contents of register 660(3),
- the contents of register 660(4),
- texture color(s),
- texture alpha(s),
- constant (register) color(s),
- constant (register) alpha(s),
- rasterized color(s),
- rasterized alpha(s),
- a few useful fixed constants,
- programmable constants,
- texture color components copied to other texture color channels (this feature is useful for dot product, intensity calculation and color space conversion),
- other inputs.

Input controls to multiplexers 656 can be specified independently for each recirculating shader stage. Multiplexer 656 outputs in example embodiment are unsigned 8-bit values or signed 10-bit values but other implementations could provide different precisions.

Example Blending Calculations/Operations

Figure 10 shows an example blending/shading operator 658. In the example embodiment, recirculating shader 602 can perform, for each stage, a computation that can be generalized as:

$$R = (D + (-1)^{sub} * ((1 - C) * A + C * B) + bias << shift$$

The arguments A, B, C and D in calculation block 664 in the example embodiment are selected from:

- four color registers,
- rasterized color (diffuse or specular),
- texture,
- the alpha components of the above colors or the defined constants,
- static constants,
- programmable constants.

Calculation block 664 operates on values A, B and C in the example embodiment. The output of calculation block 664 is passed through an optional negate block 666 and is added by adder 668 with the output of "D" multiplexer 656(D) and an optional bias value. The resulting sum can be scaled by a scaler block 670 and clamped by a clamp block 672 before being stored in any one of registers 660(1), 660(2), 660(3), 660(4) for output and/or subsequent further blending.

In the example embodiment, scale block 670 can scale by 0.5, 1, 2 or 4 – but other implementations would provide other scaling factors. Clamping block 672 can support a number of different clamping modes. In one implementation, incoming values A, B, C may be unsigned 8-bit values in one implementation, incoming value D can be a signed 10-bit value, and the output of clamp block 672 can be a signed 10-bit value.

Figure 11 shows a more detailed implementation of recirculating shader 602. In this example, calculation block 664 is implemented by hardware multipliers 664b, 664c; a "1-f" complementing block 664a; and an adder 664d. In this example, the multiplier 664b multiplies the B and C inputs and provides the resulting product to one input of adder 664d. The other multiplier 664c multiplies the A input by the value (1-C) and provides its resulting product to the other input of adder 664d. This hardware computes the equation shown in Figure 10 as being

computed by block 664. The resulting product provided by an adder 664 is outputted to the sign reversal block 666.

In the embodiment shown in Figure 11, an additional comparator 674 is provided to compare multiplexer 656 outputs A and D, with the results of the comparison being used to select between multiplexer 656 output C and a zero value (via multiplexer 676). An additional multiplexer 678 is provided in the data path to select between the output of sign operator 666 and the output of multiplexer 676 (i.e., the result of the comparison operation). Such comparisons can save recirculation stages by performing a blending calculation and a compare result operation all in one recirculation stage. This comparator 674 in the example embodiment can compare one, two or three channels simultaneously to provide 8-bit, 16-bit or 24-bit compares.

In the example embodiment shown in Figure 11, scale operation 670 provides scaling at any of the following factors:

- 0.5,
- 1,
- 2,
- 4.

In the example embodiment, different scaling blocks 670a, 670b, 670c, 670d are selected by a multiplexer 670e for providing to clamp block 672.

In the example embodiment shown in Figure 11, inputs to multiplexer 656 may optionally be passed through "color swap" blocks 680 before calculation by block 664. Color swap block 680 can be used to broadcast each color component (R, G, or B) to the other two color components. See Figures 12A, 12B. This feature can be used for dot product, intensity calculation and color space conversion, for example.

The example embodiment shown in Figure 11 includes support for static and programmable constants. In one embodiment, a constant select is used to select an arbitrary constant value determined by constant select registers specifying whether to use one of plural statically defined values or one of plural programmable color register values. Such programmable and fixed constants provide flexibility in the blending operation. In other embodiments, a smaller number of fixed constants (e.g., 1.0) could be used instead.

The Figure 11 data path can be set up for different blending operations such as, for example, those specified in D3D of DirectX 6.0 or 7.0. In the example embodiment, seven parameters are used to program a recirculating shader 602 blending stage:

- argument A,
- argument B,
- argument C,
- argument D,
- sub,
- bias,
- shift.

The following are some example blending operations:

SelectArg

$$R = (0, 0, 0, D, 0, 0, 0) = D$$

Modulate, Modulate2X, Modulate4X

$$R=(0, B, C, 0, 0, 0, 0) = B * C$$

The components of argument B and C are multiplied together. A scaling factor of 2 or 4 can be used for brightening:

$R = (0, B, C, 0, 0, 0, 1) = B * C * 2$

$R = (0, B, C, 0, 0, 0, 2) = B * C * 4$

Add

The components of the arguments are added together:

$R = (A, 0, 0, D, 0, 0, 0) = A + D$

AddSigned, AddSigned2X

The components of the arguments are added with a –0.5 bias, making the effective range of values from –0.5 to 0.5. The result can be multiplied by two for brightening:

$R = (A, 0, 0, D, 0, -0.5, 0) = A + D - 0.5$

$R = (A, 0, 0, D, 0, -0.5, 1) = (A + D - 0.5) * 2$

Subtract

The components of the second argument is subtracted from the first argument:

$R = (A, 0, 0, D, 1, -0, 0) = D - A$

AddSmooth

The basic add operation is mathematically correct for glows, fog, etc. However, it can saturate abruptly causing a contouring artifact. A better looking but less mathematically correct approach is to subtract the product:

$R = (A, 0, C, C, 0, 0, 0) = C + A * (1-C)$

BlendDiffuseAlpha, BlendTextureAlpha, BlendFactorAlpha, BlendCurrentAlpha

Linear blending is performed using the alpha from: interpolated alpha from vertices (C = rasterized alpha), alpha from current texture (C = texture alpha), a constant alpha (C = constant alpha), and/or alpha of current color (C = computed alpha):

$$R = (A, B, C_A, 0, 0, 0, 0) = A * (1-C_A) + B * C_A$$

BlendTextureAlphaPM

Linear blending with a pre-multiplied alpha:

$$R = (A, 0, C_A, D, 0, 0, 0) = D + A * (1-C_A)$$

ModulateAlpha_AddColor

The second argument is modulated with the first's alpha and the result is added to the first argument.

$$R_{GBA} = (0, B_A, C_{RGB}, B_{RGB}, 0, 0, 0) = B_{RGB} * C_{RGB}$$

ModulateColor_AddAlpha

The arguments are modulated and the first argument's alpha is then added:

$$R_{GBA} = (0, B_{RGB}, C_{RGB}, B_A, 0, 0, 0) = B_{RGB} * C_{RGB} + B_A$$

ModulateInvAlpha_AddColor

Similar to ModulateAlpha_AddColor, but it uses the inverse of the first argument's alpha:

$$R_{GBA} = (A_{RGB}, 0, C_A, C_{RGB}, 0, 0, 0) = (1 - C_A) * A_{RGB} + C_{RGB}$$

ModulateInvColor_AddAlpha

Similar to ModulateColor_AddAlpha, but it uses the inverse of the first color:

$$R_{RGB} = (A_{RGB}, 0, C_A, C_{RGB}, 0, 0, 0) = (1 - C_A) * A_{RGB} + C_{RGB}$$

ModulateInvColor_AddAlpha

Similar to ModulateColor_AddAlpha, but it uses the inverse of the first color:

$$R_{RGB} = (A_{RGB}, 0, C_{RGB}, C_A, 0, 0, 0) = (1 - C_{RGB}) * A_{RGB} + C_A$$

Specular Color and Texture

In addition to the above operation, more complicated blending can be achieved by using multiple stages. For example:

Final Color = Specular Texture * Specular Color + Diffuse Texture * Diffuse Color

It can be implemented using two stages as:

1. $R = (0, T_{SPEC}, C_{SPEC}, 0, 0, 0, 0) = T_{SPEC} * C_{SPEC}$
2. $R = (0, T_{DIFF}, C_{DIFF}, R, 0, 0, 0) = T_{SPEC} * C_{SPEC} + T_{DIFF} * C_{DIFF}$

Embossing

This example is for embossing:

Final Color = (Diffuse Color + Constant * (Normal1 – Normal2)) * Material Texture

It can be implemented using three stages as:

1. $R = (0, T_{NORM1}, Constant, C_{DIFF}, 0, 0, 0) = T_{NORM1} * Constant + C_{DIFF}$

2. 1. $R = (0, T_{NORM2}, Constant, R, 1, 0, 0) = (T_{NORM1} - T_{NORM2}) * Constant + C_{DIFF}$

3. $R = (R, 0, T_{MAT}, 0, 0, 0, 0, 0) = ((T_{NORM2}) * Constant + C_{DIFF}) * T_{MAT}$

Detail Texture

This example is for detail texturing. The difference textures have a bias of 0.5.

Final Color = Base Texture + (Difference Texture A – 0.5) + (Difference Texture B = 0.5)

It can be implemented as:

1. $R = (0, 0, 0, T_{BASE}, 0, 0, 0) = T_{BASE}$
2. $R = (T_{DIFFA}, 0, 0, R, 0, 0, 0) = T_{BASE} + T_{DIFFA} - 0.5$
3. $R = (T_{DIFFB}, 0, 0, R, 0, 0, 0) = T_{BASE} + T_{DIFFA} - 0.5 + T_{DIFFB} - 0.5$

In the example embodiment, clamp block 672 may provide any of the following clamping modes:

tev_alpha_env(mode)	clamp	description
TEV_MODE_LINEAR	TEV_CLAMP_HI TEV_CLAMP_LO	S = (R>11023) ? 1023 : (R<-1024)? –1024 : R) S + (R>255) ? 255 : ((R<0) ? 0 : R)
TEV_MODE_GE0	TEV_CLAMP_HI TEV_CLAMP_LO	S = (R >= 0) ? 255 : 0 S = (R >= 0) ? 0 : 255
	TEV_CLAMP_HI TEV_CLAMP_LO	S = (R == 0) ? 255 : 0 S = (R == 0) ? 0 : 255
	TEV_CLAMP_HI TEV_CLAMP_LO	S = (R <= 0) ? 255:0 S = (R <=0) ? 0 : 255

Alpha Function Support

The example embodiment recirculating shader 602 supports different alpha functions. Note that in the example embodiment, the alpha compare operation is not part of the recirculation stage, but rather, is performed after recirculation is complete. See Figure 11. In the example embodiment, the alpha function compares the source alpha with a reference alpha using any one of the following operations:

- always,
- never,
- not equal,
- equal,
- less,
- greater than or equal,
- less than or equal,
- greater than.

The two functions are combined in the example embodiment using:

- AND,
- OR,
- XOR,
- XNOR.

If all of the valid pixels in a pixel quad fail the alpha test, the quad is discarded and the frame buffer 702 is thus not updated. The following are some examples of what can be implemented:

Example 1

Asrc > Aref0 AND Asrc < Aref1

Example 2

Asrc > Aref0 OR Asrc < Aref1

~~The alpha functionality of recirculating shader 602 (e.g., in combination with the non-recirculating alpha compare) can be used to provide a transparency tree analogous to a shade tree. In particular, recirculating shader 602's alpha functionality can be used to provide N logical alpha operations on M alpha inputs, where N and M can be any integers. The combination of alpha compares and~~

~~alpha logical operations can be used, for example, to provide non-photorealistic~~ effects such as cartoon outlining. See, for example, ~~commonly assigned~~ copending U.S. Patent Application Serial No. _____ of Law et al. entitled "Method and Apparatus For Providing Non-Photorealistic Cartoon Outlining Within A Graphics System" (attorney docket 723-973) and its corresponding provisional application, serial no. 60/226,915, filed August 23, 2000, both of which are ~~incorporated herein by this reference.~~

Example Z Texturing

Shader 602 supports sprites with depth by modifying the screen z value using texture mapping. Once enabled, shader 602 sends four z values to pixel engine 700 per quad instead of a reference z and two slopes. Each z value is obtained by adding a z texel to the quad reference z or replacing the reference z with the z texel. See the commonly-assigned z texturing patent application referenced above.

~~Figure 13 shows an example block diagram of texture environment unit 600~~ including shader 602. Texture environment unit 600 in the example ~~embodiment~~ includes a fog operator 690 and a fog blender operator 692 ~~in~~ addition to a command section 694 and shader 602. For ~~details~~ concerning the operation of blocks 690, 692, see copending commonly assigned U.S. Patent Application Serial No. _____ of Law et al. entitled "Method And Apparatus For Providing Improved Fog Effects In A Graphics System" (attorney ref: 723-954) and its corresponding provisional application, serial no. 60/227,032, filed August 23, ~~2000, both of which are incorporated herein by this reference.~~

Figure 14 shows a more detailed block diagram of the fog operations.

Example Use of Recirculating Shader For Multitexturing

Figure 15 shows how a recirculating shader 602 can be used for multitexturing. In the example embodiment, recirculating texture unit 500 is capable of presenting a sequence of texture mapping outputs corresponding to a given surface. For example, it is possible in one implementation to map up to eight different textures onto the same primitive surface. Recirculating texture unit 500 can provide direct (and indirect) texturing operations generating a corresponding sequence of texture mapping outputs. In the example embodiment, recirculating shader 602 receives each mapped texture output as it becomes available and blends the mapped texture output with primitive surface color/alpha information derived from the lighting operations performed by transform unit 300 and/or with other, previously generated texture mappings. Recirculating shader 602 performs its blending operations in a pipelined manner so that the recirculating shader blends a texture output previously generated by texture unit 500 while the texture unit is generating a further texture output in the sequence.

In the example embodiment, recirculating shader 602 retains intermediate blending results for further blending with additional information provided by recirculating texture unit 500. Soon after recirculating texture unit 500 develops a final texture mapping output in a sequence of texture mapping outputs, recirculating shader 602 can perform a corresponding final blend operation and output the blending results via fog block 600b for depth buffering, final color blending with frame buffer 702 contents, and display.

Figure 16 shows an example multitexturing process using recirculating shader 602 shown in Figure 6. In the illustrative non-limiting example of Figure 16, transform unit 300 generates texture coordinate data (block 1002). System 50 then associates the generated texture coordinate data with a particular texture map,

and texture unit 500 retrieves the corresponding texels (texture data) from the texture map (block 1004). Meanwhile, recirculating shader 602 is configured to perform a predetermined blending/shading operation, and the retrieved texture data is provided to the recirculating shader 602 for blending (block 1008). Recirculating shader 602 can blend the retrieved texture data with some other input and/or a retained prior result (block 1010). For example, recirculating shader 602 might blend a retrieved texture data with a color or opacity value generated by lighting block 300e corresponding to a Gouraud shading operation performed on a polygon on a per-vertex basis. Blending operation 1010 might in some cases operate to blend retrieved texture data with previously retrieved texture data. Sometimes blend operation 1010 may perform a transformation on the retrieved texture data, or may simply act to pass through the retrieved texture data for storage within recirculating shader 602 for a subsequent blending operation.

Recirculating shader 602 temporarily stores the output of blend operation 1010 as an intermediate result (block 1012). The entire process may then be recirculated any number of times to retrieve and blend additional sets of texture data. In the example embodiment, recirculating shader 602 can perform blocks 1010, 1012 at the same time that texture unit 500 performs blocks 1004, 1008 to retrieve an additional texture mapping.

Figure 17 shows an example multi-texture pipeline using recirculating shader 602. Figure 17 illustrates that each time recirculating shader 602 recirculates, it provides an additional, independently controlled blending stage capable of blending a new data set with any or all of the blend results provided by previous blending operations.

Because example preferred embodiment system 50 is a real-time rendering system, the number of times that recirculating texture unit 500 can recirculate is

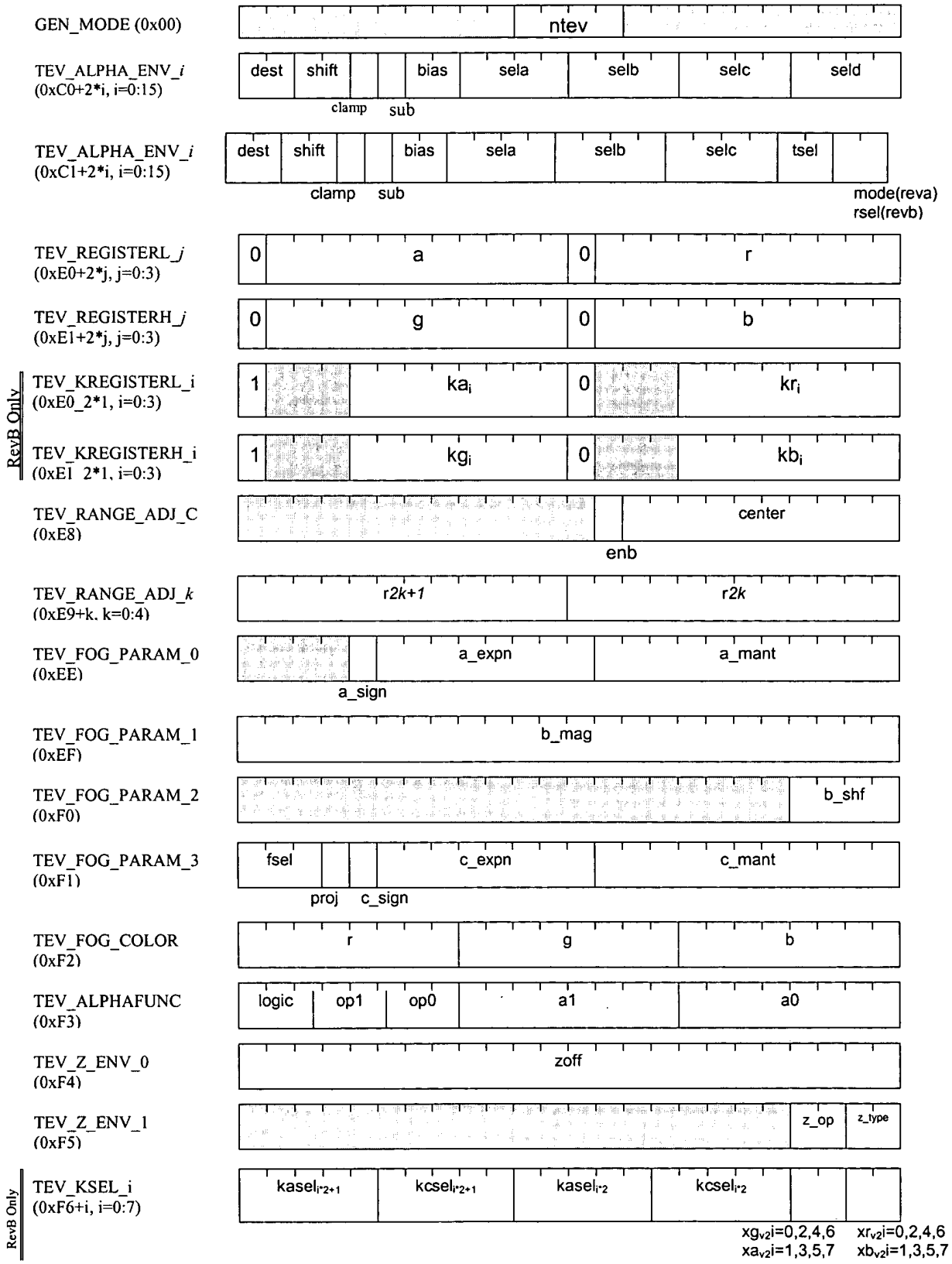
limited by the amount of time it takes for each recirculation relative to the time between mage frames (e.g., 1/30 or 1/60 of a second). In one example embodiment, the total number of recirculations that recirculating texture unit 500 can perform in a single rendering pass might be eight, although different implementations might provide different numbers of recirculations. In the example embodiment, recirculating shader 602 can recirculate approximately twice as many times as texture unit 500 can recirculate. The additional recirculations provided by recirculating shader 602 can be used to perform a number of enhanced and interesting image effects including, for example, fog, z texturing, environment mapping, embossing, detailed texturing, and other imaging effects. The texture input to recirculating shader 602 are preferably set to null during stages where texture unit 500 cannot make a texture available.

Figure 18 shows an example control step for controlling recirculating shader 602 to provide a multi-texturing operation. In this particular example, main processor 110 may specify a number of different vertices 214 to be lit and/or transformed by transform unit 300. Transform unit 300 may generate appropriate texture coordinates for application to texture unit 500, while rasterizer 400 may rasterize the vertices based on lighting calculations. The texture coordinates so generated can be used in a series of texture mapping operations based on a number of texture maps 504. These texture mapping results may be provided sequentially to a number of sequential recirculating shader 602 stages to provide multitexture blending.

Example Register Interface

Figure 19 provides detailed example definitions of register contents. The following shows further, more detailed descriptions of the various registers shown in Figure 19:

0082734.0962260


GEN_MODE (0x00)
ntev
TEV_ALPHA_ENV_i (0xC0+2*i, i=0:15)
dest
shift
clamp
sub
bias
sela
selb
selc
seld
TEV_ALPHA_ENV_i (0xC1+2*i, i=0:15)
tsel
mode(reva)
rsel(revb)
TEV_REGISTERL_j (0xE0+2*j, j=0:3)
TEV_REGISTERH_j (0xE1+2*j, j=0:3)
RevB Only
TEV_KREGISTERL_i (0xE0_2*1, i=0:3)
TEV_KREGISTERH_i (0xE1_2*1, i=0:3)
TEV_RANGE_ADJ_C (0xE8)
enb
center
TEV_RANGE_ADJ_k (0xE9+k, k=0:4)
TEV_FOG_PARAM_0 (0xEE)
a_sign
a_expn
a_mant
TEV_FOG_PARAM_1 (0xEF)
b_mag
TEV_FOG_PARAM_2 (0xF0)
b_shf
TEV_FOG_PARAM_3 (0xF1)
fsel
proj
c_sign
c_expn
c_mant
TEV_FOG_COLOR (0xF2)
TEV_ALPHAFUNC (0xF3)
logic
op1
op0
TEV_Z_ENV_0 (0xF4)
zoff
TEV_Z_ENV_1 (0xF5)
z_op
z_type
TEV_KSEL_i (0xF6+i, i=0:7)
RevB Only
xgv2i=0,2,4,6 xrv2i=0,2,4,6
xav2i=1,3,5,7 xbv2i=1,3,5,7

TO380 06/26/2006 08:47:00

register	name	format	description
gen_mode	ntev	4	Specifies the current texture stage count (1-16).
tev_color_env_*i*	dest	2	Specifies the destination register.

0	TEV_CDEST_CCO	Color Register 0
1	TEV_CDEST_CC1	Color Register 1
2	TEV_CDEST_CC2	Color Register 2
3	TEV_CDEST_CC3	Color Register 3

	shift	2

When in analog blend mode, this field specifies the amount to shift the output:

0	TEV_SHIFT_O	No shift.
1	TEV_SHIFT_1	Shift left by 1.
2	TEV_SHIFT_2	Shift left by 2.
3	TEV_SHIFT_R	Shift right by 1.

When in compare mode (bias=3, revB only), the field specifies the size and component select for the compare:

0	TEV_SHIFT_R8	Compare red channels only.
1	TEV_SHIFT_RG16	Compare red/green channels as 16 bit values.
2	TEV_SHIFT_RGB24	Compare red/green/blue channels as 24 bit values
3	TEV_SHIFT_RGB8	Compare red, green and blue channels separately

	clamp	1

Specifies the clamping operation (see Section 9).

0	TEV_CLAMP_HI	Clamp to -1024, +1023
1	TEV_CLAMP_LO	Clamp to 0.255

	sub	1

When in analog blend mode, this field specifies add or subtract of the blend result:

0	TEV_SUB_ADD	output = d + (1-c)*a + c*b + bias
1	TEV_SUB_SUB	output = d - (1-c)*a - c*b + bias

When in compare mode (bias=3, revB only), the field specifies the size and component select for the compare:

0	TEV_SUB_GT	output = d + ((a>b)?c:0)
1	TEV_SUB_EQ	output = d + ((a==b)?c:0)

	bias	2

Specifies the value of bias.

		0 TEV_SHIFT_R8 Compare red channels only. 1 TEV_SHIFT_RG16 Compare red/green channels as 16 bit values. 2 TEV_SHIFT_RGB24 Compare red/green/blue channels as 24 bit values 3 TEV_SHIFT_RGB8 Compare red, green and blue channels separately
clamp	1	Specifies the clamping operation (see Section 9). 0 TEV_CLAMP_HI Clamp to -1024, +1023 1 TEV_CLAMP_LO Clamp to 0.255
sub	1	Specifies add or subtract of the blend result. 0 TEV_SUB_ADD Add blend result. 1 TEV_CLAMP_SUB Subtract blend result. When in compare mode (bias=3, revB only), this field specifies the compare function: 0 TEV_SUB_GT output = d + ((a>b)?c:0) 1 TEV_SUB_EQ output = d + ((a==b)?c:0)
bias	2	Specifies the value of bias. 0 TEV_BIAS_ZERO 0 1 TEV_BIAS_PLUS + 0.5 2 TEV_BIAS_MINUS - 0.5 3 TEV_BIAS_COMPARE Select "compare" mode for blender. (revB only)
sela	3	Specifies argument A: 0 TEV_ASEL_A_CA0 Register 0 alpha. 1 TEV_ASEL_A_CA1 Register 1 alpha. 2 TEV_ASEL_A_CA2 Register 2 alpha. 3 TEV_ASEL_A_CA3 Register 3 alpha. 4 TEV_ASEL_A_TXA Texture alpha. 5 TEV_ASEL_A_RSA Rasterized alpha. 6 TEV_ASEL_A_KK Constant Color (see kasel) 7 TEV_ASEL_A_K00 0.0
selb	3	Specifies argument B. selb is similar to sela.
selc	3	Specifies argument C. selc is similar to sela.
seld	3	Specifies argument D. seld is similar to sela.
tsel, (revA)	2	Specifies the texture color swapping mode.

0080-6174-9600

			0 TEV_BIAS_ZERO 0 1 TEV_BIAS_PLUS + 0.5 2 TEV_BIAS_MINUS - 0.5 3 TEV_BIAS_COMPARE Select "compare" mode for blender. (revB only)
	sela	4	Specifies argument A. 0 TEV_CSEL_CC0 Register 0 color. 1 TEV_CSEL_CA0 Register 0 alpha. 2 TEV_CSEL_CC1 Register 1 color. 3 TEV_CSEL_CA1 Register 1 alpha. 4 TEV_CSEL_CC2 Register 2 color. 5 TEV_CSEL_CA2 Register 2 alpha. 6 TEV_CSEL_CC3 Register 3 color. 7 TEV_CSEL_CA3 Register 3 alpha. 8 TEV_CSEL_TXC Texture color. 9 TEV_CSEL_TXA Texture alpha. A TEV_CSEL_RSC Rasterized color. B TEV_CSEL_RSA Rasterized alpha. C REVA: TEV_CSEL_K10 1.0 REVB: TEV_CSEL-KK Constant Color (see kcsel) D TEV_CSEL_K05 0.5 E TEV_CSEL_K25 0.25 F TEV_CSEL_K00 0.0
	selb	4	Specifies argument B. selb is similar to sela
	selc	4	Specifies argument C. selc is similar to sela.
	seld	4	Specifies argument D. seld is similar to sela.
tev_alpha_env_*i*	dest	2	Specifies the destination register. 0 TEV_ADEST_CA0 Color Register 0 1 TEV_ADEST_CA1 Color Register 1 2 TEV_ADEST_CA2 Color Register 2 3 TEV_ADEST_CA3 Color Register 3
	shift	2	Specifies the amount to shift. 0 TEV_SHIFT_0 No shift. 1 TEV_SHIFT_1 Shift left by 1. 2 TEV_SHIFT_2 Shift left by 2. When in compare mode (bias=3, revB only), the field specifies the size and component select for the compare:

DOCKET ALARM

			0 TEV_SWAP_0 RGBA => RGBA 1 TEV_SWAP_R RGBA => RRRA 2 TEV_SWAP_G RGB1 => GGGA 3 TEV_SWAP_B RGBA => BBBA
	mode (revA)	2	Specifies the clamping mode (see Section 9). Rev. A. only! 0 TEV_MODE_LINEAR Linear clamping. 1 TEV_MODE_GE0 Greater than ore equal to 0. 2 TEV_MODE_EQ0 Equal to 0. 3 TEV_MODE_LE0 Less than or equal to 0.
	tsel, rsel (revB)	2	Specifies the texture and raster color swapping mode. 0 TEV_SWAP_0 Use swap mode 0 1 TEV_SWAP_R Use swap mode 1 2 TEV_SWAP_G Use swap mode 2 3 TEV_SWAP_B Use swap mode 3
tev_registerl_*i*	r, a	s2.8	Specifies the value of the texture current color.
tev_registerh_*i*	g, b	s2.8	Specifies the value of the texture current color.
tev_kregisterl_*i*	kr, ka	0.8	Specifies the value of the constant color. This feature only applies to rev B.
tev_kregisterh_*i*	kg, kb	0.8	Specifies the value of the constant color. This feature only applies to rev B.
tev_range_adj_c	center	10	Specifies the screen's x center for range adjustment.
	enb	1	Enable range adjustment 0 TEV_ENB_DISABLE Disable range adjustment. 1 TEV_ENB_ENABLE Enable range adjustment.
rev_range_adj_*k*	r*2k*, r*2k+1*	u4.8	Specifies the range adjustment function. $adj = \frac{\sqrt{x^2 + k^2}}{k}$
tev_fog_param_0	a	s11e8	Specifies the "a" parameter of the screen to eye space conversion function: $Z_e = \frac{a}{b - Z_s}$
tev_fog_param_1	b_mag	u0.24	Specifies the "b" parameter of the z screen to eye space conversion function: $Z_e = \frac{a}{b_mag - (zs >> b_shf)}$
tev_fog_param_2	b_shf	5	Specifies the amount to pre-shift screen z. This is equivalent to the value of "b" parameter's exponent +1.

<table>
<tr><td>tev_fog_param_3</td><td>fsel</td><td>3</td><td>Specifies the fog type as follows:
<table><tr><td>0</td><td>TEV_FSEL_OFF</td><td>No Fog.</td></tr><tr><td>1</td><td>reserved</td><td></td></tr><tr><td>2</td><td>TEV_FSEL_LIN</td><td>Exponential Fog</td></tr><tr><td>3</td><td>reserved</td><td></td></tr><tr><td>4</td><td>TEV_FSEL_EXP</td><td>Exponential Fog</td></tr><tr><td>5</td><td>TEV_FSEL_EX2</td><td>Exponential Squared Fog</td></tr><tr><td>6</td><td>TEV_FSEL_BXP</td><td>Backward Exp Fog</td></tr><tr><td>7</td><td>TEV_FSEL_BX2</td><td>Backward Exp Squared Fog</td></tr></table></td></tr>
<tr><td></td><td>proj</td><td>1</td><td>Specifies whether we have a perspective or orthographic projection.
<table><tr><td>0</td><td>TEV_FOG_PERSP</td><td>Perspective projection</td></tr><tr><td>1</td><td>TEV_FOG_ORTHO</td><td>Orthographic projection</td></tr></table></td></tr>
<tr><td></td><td>c</td><td>s1 1e8</td><td>Specifies the amount to subtract from eye-space Z after range adjustment.</td></tr>
<tr><td>tev_fog_color</td><td>r, g, b</td><td>8</td><td>Specifies the value of fog color.</td></tr>
<tr><td>tev_alphafunc</td><td>op0</td><td>3</td><td>Specifies under what condition the alpha 0 for a pixel is to be forced to 1.
<table><tr><td>0</td><td>TEV_AOP_NEVER</td><td>Never</td></tr><tr><td>1</td><td>TEV_AOP_LESS</td><td>Alpha < AF_VAL</td></tr><tr><td>2</td><td>TEV_AOP_EQUAL</td><td>Alpha = AF_VAL</td></tr><tr><td>3</td><td>TEV_AOP_LE</td><td>Alpha <= AF_VAL</td></tr><tr><td>4</td><td>TEV_AOP_GREATER</td><td>Alpha > AF_VAL</td></tr><tr><td>5</td><td>TEV_AOP_NOTEQUAL</td><td>Alpha != AF_VAL</td></tr><tr><td>6</td><td>TEV_AOP_GE</td><td>Alpha >= AF_VAL</td></tr><tr><td>7</td><td>TEV_AOP_ALWAYS</td><td>Always</td></tr></table></td></tr>
<tr><td></td><td>op1</td><td>3</td><td>Specifies alpha operation 1. Similar to op0.</td></tr>
<tr><td></td><td>logic</td><td>2</td><td>Specifies the logic operation in combining the two alpha comparison.
<table><tr><td>0</td><td>TEV_LOGIC_AND</td><td>AND</td></tr><tr><td>1</td><td>TEV_LOGIC_OR</td><td>OR</td></tr><tr><td>2</td><td>TEV_LOGIC_XOR</td><td>XOR</td></tr><tr><td>3</td><td>TEV_LOGIC_XNOR</td><td>XNOR</td></tr></table></td></tr>
<tr><td></td><td>a0</td><td>8</td><td>Reference alpha 0.</td></tr>
<tr><td></td><td>a1</td><td>8</td><td>Reference alpha 1.</td></tr>
<tr><td>tev_env_z_0</td><td>zoff</td><td>u24.0</td><td>Specifies the z bias used in a z texture.</td></tr>
</table>

0080-3434-US-P1

tev_env_z_1	type	2	Specifies the z texel type.

0	TEV_Z_TYPE_U8	u8.0
1	TEV_Z_TYPE_U16	u16.0
2	TEV_Z_TYPE_U24	u24.0
3	reserved	

	op	2	Enables z texturing.

0	TEV_Z_OP_OFF	Disable
1	TEV_Z_OP_ADD	Add
2	TEV_Z_OP_REP	Replace
3	reserved	

tev_ksel_i (RevB)	kcsel, kasel	5	Selects constant color/scalar for each of 16 states. This feature only applies to rev B.

0	TEV_KSEL_1	1.0
1	TEV_KSEL_7_8	0.875
2	TEV_KSEL_3_4	0.75
3	TEV_KSEL_5_8	0.625
4	TEV_KSEL_1_2	0.5
5	TEV_KSEL_3_8	0.375
6	TEV_KSEL_1_4	.025
7	TEV_KSEL_1_8	0.125
8		
9		
10		
11		
16	TEV_KSEL_K0_R	Constant Color 0 (r)
17	TEV_KSEL_K1_R	Constant Color 1 (r)
18	TEV_KSEL_K2_R	Constant Color 2 (r)
19	TEV_KSEL_K3_R	Constant Color 3 (r)
20	TEV_KSEL_K0_R	Constant Color 0 (g)
21	TEV_KSEL_K1_G	Constant Color 1 (g)
22	TEV_KSEL_K2_G	Constant Color 2 (g)
23	TEV_KSEL_K3_G	Constant Color 3 (g)
24	TEV_KSEL_K0_B	Constant Color 0 (b)
25	TEV_KSEL_K1_B	Constant Color 1 (b)
26	TEV_KSEL_K2_B	Constant Color 2 (b)
27	TEV_KSEL_K3_B	Constant Color 3 (b)
28	TEV_KSEL_K0_A	Constant Color 0 (a)
29	TEV_KSEL_K1_A	Constant Color 1 (a)
30	TEV_KSEL_K2_A	Constant Color 2 (a)
31	TEV_KSEL_K3_A	Constant Color 3 (a)

kcsel only

	xr, xg,		Specifies one of four swap modes. During each stage, "tsel" selects the swap mode for texture and "rsel" selects the swap

0082474-960529

xb,xa

mode for the rasterization color:

red		green		blue		alpha	
xr		xg		xb		xa	
0 0	red	0 0	red	0 0	red	0 0	red
0 1	green	0 1	green	0 1	green	0 1	green
1 0	blue	1 0	blue	1 0	blue	1 0	blue
1 1	alpha	1 1	alpha	1 1	alpha	1 1	alpha

On reset, these values get initialized as follows (for revA compatibility):

	xr	xg	xb	xa
Swap Mode 0	0 0	0 1	1 0	1 1
Swap Mode 1	0 0	0 0	0 0	1 1
Swap Mode 2	0 1	0 1	0 1	1 1
Swap Mode 3	1 0	1 0	1 0	1 1

D02020 .0080.PDF

The following are example application programming interface calls:

GXSetTevOp

Description: This is a convenience function designed to make initial programming of the Texture Environment unit easier. This macro calls GXSetTevColorIn, GXSetTevColorOp, GXSetTevAlphaIn, and GXSetTevAlphaOp with predefined arguments to implement familiar texture combining functions.

To enable a consecutive set of recirculating shader stages, the application should call the GXSetNumTevStages function.

In the table below, Cv is the output color for the stage, Cr is the output color of previous stage, and Ct is the texture color. Av is the output alpha for a stage, Ar is the output alpha of previous stage, and At is the texture alpha. As a special case, rasterized color (GX_CC_RASC) is used as Cr and rasterized alpha (GX_CA_RASA) is used as Ar at the first recirculating shader stage because there is no previous stage.

Mode	Color Op	Alpha Op
GX_MODULATE	Cv=CrCt	Av=ArAt
GX_DECAL	Cv=(1-At)Cr + AtCt	Av=Ar
GX_REPLACE	Cv=Ct	Av=At
GX_BLEND	Cv=(1-Ct)Cr + Ct	Av=AtAr
GX_PASSCLR	Cv=Cr	Av=Ar

Arguments: id=stage id, mode=predefined color combining modes.

Example usage:
void GXSetTevOp(GXTevStageID id, GXTevMode mode);

GXTevStageID

Enumerated Values

GX_TEVSTAGE0
GX_TEVSTAGE1
GX_TEVSTAGE2
GX_TEVSTAGE3
GX_TEVSTAGE4
GX_TEVSTAGE5
GX_TEVSTAGE6
GX_TEVSTAGE7
GX_TEVSTAGE8
GX_TEVSTAGE9
GX_TEVSTAGE10
GX_TEVSTAGE11
GX_TEVSTAGE12
GX_TEVSTAGE13
GX_TEVSTAGE14
GX_TEVSTAGE15
GX_MAX_TEVSTAGE

Description
Texture Environment (Recirculating shader) stage name.

GXTevMode

Enumerated Values:
GX_DECAL
GX_MODULATE
GX_REPLACE
GX_PASSCLR
GX_BLEND

Description: sets Texture Environment control.

GXSetNumTevStages

Description

This function enables a consecutive number of Texture Environment (recirculating shader) stages. The output pixel color (before fogging and blending) is the result from the last stage. The last recirculating shader stage must write to register GX_TEVPREV, see GXSetTevColorOp and GXSetTevAlphaOp. At least one recirculating shader stage should be enabled. If a Z-texture is enabled, the Z texture is looked up on the last stage, see GXSetZTexture.

The association of lighting colors, texture coordinates, and texture maps with a recirculating shader stage is set using GXSetTevOrder. The number of texture coordinates available is set using GXSetNumTexGens. The number of color channels available is set using GXSetNumChans.

GXInit will set nStages to 1 as a default.

Arguments: nStages
Number of active recirculating shader stages. Minimum value is 1, maximum value is 16
Example usage: void GXSetNumTevStages(u8 nStages);

GXSetTevColorIn

Description

This function sets the input operands of the Texture Environment (recirculating shader) color combiner unit. The input operands a, b, and c are RGB colors where each component is unsigned 8-bit (0 <= a,b,c <= 255). The d input operand is an RGB color where each component is a signed 10-bit input (-1024 <= d <= 1023).

In the cases where the input operand is an alpha value (GX_CC_A0, GX_CC_A1, GX_CC_A2, GX_CC_APREV, GX_CC_TEXA, GX_CC_RASA), the alpha value is replicated across the three color channels (R=A, G=A, B=A).

The function implemented by this recirculating shader stage is set using the function GXSetTevColorOp.

The output of this stage is directed by default to register GX_TEVPREV (see GXInit), but may be set explicitly by GXSetTevColorOp.

The registers used to store the output of Recirculating shader stages can also be used as inputs, GX_CC_C0, GX_CC_C1, GX_CC_C2, GX_CC_CPREV. You can program these registers with constant color values using GXSetTevColor or GXSetTevColorS10.

Each register can store either an unsigned 8-bit number or a signed 10-bit number per component (RGB). If a signed 10-bit number is selected for inputs a, b, or c, the number is truncated to 8 bits. No attempt is made to convert the number, the most significant bits are simply discarded.

The input operands GX_CC_RASC and GX_CC_RASA are the result of the per-vertex lighting equations. The input operands GX_CC_TEXC and GX_CC_TEXA are the texture inputs for this stage. The texture color input GX_CC_TEXC, may have its color components swapped before input by setting operands GX_TC_TEXRRR, GX_TC_TEXGGG or GX_TC_TEXBBB. You can select one of the swap operands per Recirculating shader stage. In an example embodiment, it is illegal to use both GX_TC_TEXRRR and GX_TC_GGG in the same stage.

GXSetTevOrder associates a shader stage with particular colors and textures.

Arguments

stage	Name of the Recirculating shader stage.
a	Input color operand, unsigned 8b per component.
b	Input color operand, unsigned 8b per component.
c	Input color operand, unsigned 8b per component.
d	Input color operand, signed 10b per component

Example usage:

```
void GXSetTevColorIn(
        GXTevStageID  stage,
        GXTevColorArg  a,
        GXTevColorArg  b,
        GXTevColorArg  c,
        GXTevColorArg  d );
```

GXSetTevAlphaIn

Description

This function sets the input operands for one stage of the Texture Environment (recirculating shader) alpha combiner unit. The input operands a, b, and c are unsigned 8-bit inputs (0 <= a,b,c <= 255). The d input operand is a signed 10-bit input (-1024 <= d <= 1023).

Each shader stage implements the following function:

reg = (d (op) ((1.0 - c)*a + c*b) + bias) * scale;

The operations described by op, bias, and scale are programmable using the GXSetTevAlphaOp function.

The output of this stage is directed by default to register GX_TEVPASS (see GXInit), but may be set explicitly by GXSetTevAlphaOp. The result can be clamped to two ranges, 0 to 255 or -1024 to 1023, based on the clamp mode set by

GXSetTevClampMode. When the input a, b, or c is from a signed 10-bit number (either the results of a previous recirculating shader stage or an input constant) only the 8 least-significant bits are used. There is no attempt to convert the number, the upper bits are simply discarded.

The registers used to store the output of Recirculating shader stages can also be used as inputs, GX_CA_A0, GX_CA_A1, GX_CA_A2, GX_CA_APREV. You can program these registers with constant alpha values using GXSetTevColor or GXSetTevColorS10.

The input operand GX_CA_RASA is the result of the per-vertex lighting equations. The input operand GX_CA_TEXA is the texture alpha input for this stage. You can select the colors and textures to which these inputs correspond using GXSetTevOrder.

Arguments

stage	The name of the stage.
a	Input operand, u8.
b	Input operand, u8.
c	Input operand, u8
d	Input operand, s10.

Example usage:

```
void GXSetTevAlphaIn(
   GXTevStageID  stage,
   GXTevAlphaArg  a,
   GXTevAlphaArg  b,
   GXTevAlphaArg  c,
   GXTevAlphaArg  d );
```

GXSetTevColorOp

Description

This function sets the op, scale, bias, and clamping operation for the color combiner function for this stage of the Texture Environment (recirculating shader) unit. This function also specifies the output register, out_reg, that will contain the result of the color combiner function. The color combiner function is:

out_reg = (d (op) ((1.0 - c)*a + c*b) + bias) * scale;

The input parameters a, b, c, and d are selected using the GXSetTevColorIn function. The a, b, and c inputs are unsigned 8b inputs (0 <= a,b,c <= 255). The d input is a signed 10b input (-1024 <= d <= 1023). The result, out_reg, can also be a signed 10b result, depending on the clamp enable and the current clamping mode, see GXSetTevClampMode.

The recirculating shader output registers are shared among all the recirculating shader stages. The recirculating shader output registers can also be used as constant color inputs, so the application should be careful to allocate input and output registers so no collision occurs when implementing a particular equation. The application must output to GX_TEVPREV in the last active recirculating shader stage.

The function GXSetTevOp provides a simpler way to set the parameters of GXSetTevColorIn and GXSetTevColorOp based on predefined equation names. You should not mix usage of GXSetTevOp and GXSetTevColorIn/GXSetTevColorOp.

GXSetTevOp makes some assumptions about the output register usage, namely that GX_TEVPREV is always the output register and is used to pass the result of the previous recirculating shader stage to the next recirculating shader stage.

Arguments

stage	Recirculating shader stage name.
op	Recirculating shader operation.
add_bias scale	Bias value. Scale value.
clamp	Clamp results when GX_TRUE
out_reg	Output register name. The last active Recirculating shader stage writes to GX_TEVPREV.

Example usage:

```
void GXSetTevColorOp(
        GXTevStageID   stage,
        GXTevOp        op,
        GXTevBias      bias,
        GXTevScale     scale,
        GXBool         clamp,
        GXTevRegID     out_reg );
```

GXSetTevAlphaOp

Description

This function sets the op, scale, bias, and clamping operation for the alpha combiner function for this stage of the Texture Environment (recirculating shader) unit. This function also specifies the register, out_reg, that will contain the result of the alpha combiner function. The alpha combiner function is:

out_reg = (d (op) ((1.0 - c)*a + c*b) + bias) * scale;

The input parameters a, b, c, and d are set using GXSetTevAlphaIn. The a, b, and c inputs are unsigned 8b inputs (0<= a,b,c <= 255). The d input is a signed 10b input (-1024 <= d <= 1023). The result, out_reg, can also be a signed 10b result, depending on the clamp enable and the current clamping mode, see GXSetTevClampMode.

You must enable a consecutive number of recirculating shader stages using GXSetTevStages. The last active recirculating shader stage writes its output to register GX_TEVPREV.

Arguments

stage	The name of the recirculating shader stage.
op	recirculating shader operation.
bias	Bias value.
scale	Scale value.
clamp	Clamp results when GX_TRUE.
out_reg	Output register name.

Example usage:

```
void GXSetTevAlphaOp(
        GXTevStageID    stage,
        GXTevOp         op,
        GXTevBias       bias,
        GXTevScale      scale,
        GXBool          clamp,
        GXTevRegID      out_reg );
```

GXSetTevColor

Description

This function is used to set one of the constant color registers in the Texture Environment (recirculating shader) unit. These registers are available to all recirculating shader stages. At least one of these registers is used to pass the output of one recirculating shader stage to the next in a multi-texture configuration. The application is responsible for allocating these registers so that no collisions in usage occur.

This function can set unsigned 8-bit colors. To set signed, 10-bit colors use GXSetTexColorS10.

Arguments

id = Color register id.
color= Constant color value.

Example usage: void GXSetTevColor(GXTevRegID id, GXColor color);

GXSetTevColorS10

Description

This function is used to set one of the constant color registers in the Texture Environment (recirculating shader) unit. These registers are available to all recirculating shader stages. At least one of these registers is used to pass the output of one recirculating shader stage to the next in a multi-texture configuration. The application is responsible for allocating these registers so that no collisions in usage occur.

This function enables the color components to be signed 10-bit numbers. To set 8-bit unsigned colors (the common case) use GXSetTevColor.

Arguments

id= Color register id.
color=Constant color value. Each color component can have the range -1024 to +1023.

Example usage: void GXSetTevColorS10(GXTevRegID id, GXColorS10 color);

GXSetTevClampMode

Description

This function sets the clamp mode for this stage in the Texture Environment (recirculating shader) unit. This mode is used for both alpha and color combiners. The mode effects how the clamp controls set by GXSetTevColorOp and GXSetTevAlphaOp are interpreted as shown in the table below. R is the TEV stage result color.

GXInit sets mode to GX_TC_LINEAR.

mode - shared for alpha and color TEV	clamp - independent for alpha and color TEV	description
GX_TC_LINEAR	GX_FALSE GX_TRUE	clamp such that -1024 <= R <= 1023 clamp such that 0 <= R <= 255
GX_TC_GE	GX_FALSE GX_TRUE	output = (R >= 0) ? 255 : 0 output = (R >= 0) ? 0 : 255
GX_TC_EQ	GX_FALSE GX_TRUE	output = (R == 0) ? 255 : 0 output = (R == 0) ? 0 : 255
GX_TC_LE	GX_FALSE GX_TRUE	output = (R <= 0) ? 255 : 0 output = (R <= 0) ? 0 : 255

stage=Tev stage ID.
mode=Clamp mode (Accepted values are GX_TC_LINEAR, GX_TC_GE, GX_TC_LE, GX_TC_EQ).

Example usage:

```
void GXSetTevClampMode(
        GXTevStageID    stage,
        GXTevClampMode  mode );
```

GXSetAlphaCompare

Description

This function sets the parameters for the alpha compare function which uses the alpha output from the last active Texture Environment (recirculating shader) stage. The number of active recirculating shader stages are specified using GXSetTevStages.

The output alpha can be used in the blending equation (see GXSetBlendMode) to control how source and destination (frame buffer) pixels are combined.

The alpha compare operation is:

alpha_pass = (alpha_src (comp0) ref0) (op) (alpha_src (comp1) ref1)

where alpha_src is the alpha from the last active Recirculating shader stage. As an example, you can implement these equations:

alpha_pass = (alpha_src > ref0) AND (alpha_src < ref1)

or

alpha_pass = (alpha_src > ref0) OR (alpha_src < ref1)

The Z compare can occur either before or after texturing, see GXSetZCompLoc. In the case where Z compare occurs before texturing, the Z is written based only on the Z test. The color is written if both the Z test and alpha test pass.

When Z compare occurs after texturing, the color and Z are written if both the Z test and alpha test pass. When using texture to make cutout shapes (like billboard trees) that need to be correctly Z buffered, you should configure the pipeline to Z buffer after texturing.

Arguments

comp0=Comparison subfunction 0.

ref0=Reference value for subfunction 0, 8-bit.
op=Operation for combining subfunction0 and subfunction1. Accepted values are: GX_AOP_AND, GX_AOP_OR,
GX_AOP_XOR, GX_AOP_XNOR.
comp1=Comparison subfunction 1.
ref1=Reference value for subfunction 1, 8-bit.

Example usage:

```
void GXSetAlphaCompare(
        GXCompare   comp0,
        u8          ref0,
        GXAlphaOp   op,
        GXCompare   comp1,
        u8          ref1 );
```

GXSetTevOrder

Description

This function specifies the texture and rasterized color that will be available as inputs to this Texture Environment (recirculating shader) stage. The texture coordinate coord is generated from input attributes using the GXSetTexCoordGen function, and is used to look up the texture map, previously loaded by GXLoadTexObj. The color to rasterize for this stage is also specified. The color is the result of per-vertex lighting which is controlled by the GXSetChanCtrl function.

Note that this function does not enable the recirculating shader stage. To enable a consecutive number of Recirculating shader stages, starting at stage GX_TEVSTAGE0, use the GXSetNumTevStages function.

The operation of each recirculating shader stage is independent. The color operations are controlled by GXSetTevColorIn and GXSetTevColorOp. The alpha operations are controlled by GXSetTevAlphaIn and GXSetTevAlphaOp.

The number of texture coordinates available for all the active recirculating shader stages is set using GXSetNumTexGens. The number of color channels available for all the active recirculating shader stages is set using GXSetNumChans. Active recirculating shader stages should not reference more texture coordinates or colors than are being generated.

Using GXSetTevOrder, it is possible to broadcast a single texture coordinate to many textures as long as the textures are the same size:

```
GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD0, GX_TEXMAP0, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE1, GX_TEXCOORD0, GX_TEXMAP1, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE2, GX_TEXCOORD0, GX_TEXMAP2, GX_COLOR1A1);
GXSetTevOrder(GX_TEVSTAGE3, GX_TEXCOORD0, GX_TEXMAP3, GX_COLOR0A0);
```

You may also use any generated texture coordinate in any recirculating shader stage:

```
GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD3, GX_TEXMAP0, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE1, GX_TEXCOORD2, GX_TEXMAP1, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE2, GX_TEXCOORD1, GX_TEXMAP2, GX_COLOR1A1);
GXSetTevOrder(GX_TEVSTAGE3, GX_TEXCOORD0, GX_TEXMAP3, GX_COLOR0A0);
```

If no texture is used in a recirculating shader stage, set coord and map to NULL:

```
GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD2, GX_TEXMAP0, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE1, GX_TEXCOORD_NULL, GX_TEXMAP_NULL,
GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE2, GX_TEXCOORD1, GX_TEXMAP2, GX_COLOR1A1);
GXSetTevOrder(GX_TEVSTAGE3, GX_TEXCOORD0, GX_TEXMAP3, GX_COLOR0A0);
```

If no color is used in a recirculating shader stage, set color to NULL:

```
GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD3, GX_TEXMAP0, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE1, GX_TEXCOORD2, GX_TEXMAP1,
GX_COLOR_NULL);
GXSetTevOrder(GX_TEVSTAGE2, GX_TEXCOORD1, GX_TEXMAP2, GX_COLOR1A1);
GXSetTevOrder(GX_TEVSTAGE3, GX_TEXCOORD0, GX_TEXMAP3, GX_COLOR0A0);
```

GXSetTevOrder will scale the normalized texture coordinates produced by GXSetTexCoordGen according to the size of the texture map in the function call. For this reason, texture coordinates can only be broadcast to multiple texture maps if and only if the maps are the same size. In some case, you may want to generate a texture coordinate having a certain scale, but disable the texture lookup (this comes up when generating texture coordinates for indirect bump mapping). To accomplish this, use the GX_TEXMAP_DISABLE flag:

```
GXSetTevOrder(GX_TEVSTAGE1, GX_TEXCOORD0, GX_TEXMAP3 |
GX_TEXMAP_DISABLE, GX_COLOR_NULL);
```

This will scale GX_TEXCOOR0 using GX_TEXMAP3 but disable the lookup of GX_TEXMAP3.

GXInit describes the default recirculating shader order.

Arguments

stage=Recirculating shader stage ID.
coord=Texture coordinate ID.
map=Texture Map ID.
color =Color channel. Accepted values are: GX_COLOR0A0, GX_COLOR1A1 and GX_COLOR_NULL.

Example usage:

```
void GXSetTevOrder(
   GXTevStageID   stage,
   GXTexCoordID   coord,
   GXTexMapID     map,
   GXChannelID    color
);
```


0068743.0960

Examples:

This page shows some samples of the Texture Environment (TEV) settings.

One Rasterized Color

e.g. Polygons with vertex colors
e.g. Vertex lighting

This configuration passes rasterized color channel directly by using PASSCLR operation. No texture is used.

```
// One Rasterized Color
// The channel COLOR0A0 is supposed to have lit color.

GXSetNumTevStages(1);
GXSetTevOrder(
   GX_TEVSTAGE0,
   GX_TEXCOORD_NULL,
   GX_TEXMAP_NULL,
   GX_COLOR0A0 );
GXSetTevOp(GX_TEVSTAGE0, GX_PASSCLR);
```

One Texture

e.g. Simple texture mapping

This configuration is used for displaying the texture color directly. No rasterized color can be used.

```
// One Texture
// A texture should be loaded to GX_TEXMAP0.
// An appropriate texcoord generation should be set to GX_TEXCOORD0.

GXSetNumTevStages(1);
GXSetTevOrder(
   GX_TEVSTAGE0,
   GX_TEXCOORD0,
   GX_TEXMAP0,
   GX_COLOR_NULL );
```


0080-24-NGC-US

```
GXSetTevOp(GX_TEVSTAGE0, GX_REPLACE);
```

One Texture Modulated by Rasterized Color

e.g. Lit material texture

This configuration uses the MODULATE operation.

```
// One Texture Modulated by Rasterized Color
// The channel COLOR0A0 is supporsed to have lit color.
// A texture should be loaded to GX_TEXMAP0.
// An appropriate texcoord generation should be set to GX_TEXCOORD0.

GXSetNumTevStages(1);
GXSetTevOrder(
  GX_TEVSTAGE0,
  GX_TEXCOORD0,
  GX_TEXMAP0,
  GX_COLOR0A0 );
GXSetTevOp(GX_TEVSTAGE0, GX_MODULATE);
```

One Texture Overlayed on Rasterized Color

e.g. Highlight map on diffuse lit surface
e.g. Projected shadow map on lit surface

This configuration uses the DECAL operation. The texture should contain alpha value which will be used for blending.

```
// One Texture Modulated by Rasterized Color
// The channel COLOR0A0 is supporsed to have lit color.
// A texture should be loaded to GX_TEXMAP0.
// An appropriate texcoord generation should be set to GX_TEXCOORD0.

GXSetNumTevStages(1);
GXSetTevOrder(
  GX_TEVSTAGE0,
  GX_TEXCOORD0,
```

```
    GX_TEXMAP0,
    GX_COLOR0A0 );
GXSetTevOp(GX_TEVSTAGE0, GX_DECAL);
```

Constant Color

This configuration uses neither the output from lighting unit or any texture.

```
// Constant color from TEV register

GXSetNumTevStages(1);
GXSetTevOrder(
    GX_TEVSTAGE0,
    GX_TEXCOORD_NULL,
    GX_TEXMAP_NULL,
    GX_COLOR_NULL );
GXSetTevColorIn(  // output = Register0
    GX_TEVSTAGE0,
    GX_CC_ZERO,
    GX_CC_ZERO,
    GX_CC_ZERO,
    GX_CC_C0 );
GXSetTevColorOp(
    GX_TEVSTAGE0,
    GX_TEV_ADD,
    GX_TB_ZERO,
    GX_CS_SCALE_1,
    GX_DISABLE,
    GX_TEVPREV );
GXSetTevColor(GX_TEVREG0, constColor);
```

Add Two Rasterized Colors

e.g. Diffuse lit color + Specular lit color

No texture is used. The first stage passes the first rasterized color by using PASSCLR operation. The second stage adds two colors where a detailed setting is required.

```
// Add Two Rasterized Colors
// Two Color channels COLOR0/COLOR1 will be used.

GXSetNumTevStages(2);

// Stage0 simply passes the rasterized color.
GXSetTevOrder(
  GX_TEVSTAGE0,
  GX_TEXCOORD_NULL,
  GX_TEXMAP_NULL,
  GX_COLOR0A0 );
GXSetTevOp(GX_TEVSTAGE0, GX_PASSCLR);

// Stage1 adds the second color and output from previous stage.
GXSetTevOrder(
  GX_TEVSTAGE1,
  GX_TEXCOORD_NULL,
  GX_TEXMAP_NULL,
  GX_COLOR0A0);
GXSetTevColorIn(  // output = RASC + CPREV
  GX_TEVSTAGE1,
  GX_CC_ZERO,
  GX_CC_RASC,
  GX_CC_ONE,
  GX_CC_CPREV );
GXSetTevColorOp(
  GX_TEVSTAGE1,
  GX_TEV_ADD,
  GX_TB_ZERO,
  GX_CS_SCALE_1,
  GX_ENABLE,
  GX_TEVPREV );
GXSetTevClampMode(GX_TEVSTAGE1, GX_TC_LINEAR);

Add Rasterized Color and Alpha

e.g. Diffuse lit color + Specular lit color processed in alpha channel
```

If the specular color is allowed to be white only, you may use alpha channel for specular lit color which will be broadcasted to each RGB component on a TEV stage. Since it requires only one stage, we can obtain better fill-rate than using two channels. This method can be used if the alpha is not reserved for another purpose.

```
// Add Rasterized Color and Alpha
// Color0/Alpha0 may be processed independently.

GXSetNumTevStages(1);
GXSetTevOrder(
  GX_TEVSTAGE0,
  GX_TEXCOORD_NULL,
  GX_TEXMAP_NULL,
  GX_COLOR0A0 );
GXSetTevColorIn(  // output = RASC + RASA
  GX_TEVSTAGE0,
  GX_CC_ZERO,
  GX_CC_RASA,
  GX_CC_ONE,
  GX_CC_RASC );
GXSetTevColorOp(
  GX_TEVSTAGE0,
  GX_TEV_ADD,
  GX_TB_ZERO,
  GX_CS_SCALE_1,
  GX_ENABLE,
  GX_TEVPREV );
GXSetTevClampMode(GX_TEVSTAGE0, GX_TC_LINEAR);
```

Other Example Compatible Implementations

Certain of the above-described system components 50 could be implemented as other than the home video game console configuration described above. For example, one could run graphics application or other software written for system 50 on a platform with a different configuration that emulates system 50

or is otherwise compatible with it. If the other platform can successfully emulate, simulate and/or provide some or all of the hardware and software resources of system 50, then the other platform will be able to successfully execute the software.

As one example, an emulator may provide a hardware and/or software configuration (platform) that is different from the hardware and/or software configuration (platform) of system 50. The emulator system might include software and/or hardware components that emulate or simulate some or all of hardware and/or software components of the system for which the application software was written. For example, the emulator system could comprise a general purpose digital computer such as a personal computer, which executes a software emulator program that simulates the hardware and/or firmware of system 50.

Some general purpose digital computers (e.g., IBM or MacIntosh personal computers and compatibles) are now equipped with 3D graphics cards that provide 3D graphics pipelines compliant with DirectX or other standard 3D graphics command APIs. They may also be equipped with stereophonic sound cards that provide high quality stereophonic sound based on a standard set of sound commands. Such multimedia-hardware-equipped personal computers running emulator software may have sufficient performance to approximate the graphics and sound performance of system 50. Emulator software controls the hardware resources on the personal computer platform to simulate the processing, 3D graphics, sound, peripheral and other capabilities of the home video game console platform for which the game programmer wrote the game software.

Figure 20A illustrates an example overall emulation process using a host platform 1201, an emulator component 1303, and a game software executable binary image provided on a storage medium 62. Host 1201 may be a general or

special purpose digital computing device such as, for example, a personal computer, a video game console, or any other platform with sufficient computing power. Emulator 1303 may be software and/or hardware that runs on host platform 1201, and provides a real-time conversion of commands, data and other information from storage medium 62 into a form that can be processed by host 1201. For example, emulator 1303 fetches "source" binary-image program instructions intended for execution by system 50 from storage medium 62 and converts these program instructions to a target format that can be executed or otherwise processed by host 1201.

As one example, in the case where the software is written for execution on a platform using an IBM PowerPC or other specific processor and the host 1201 is a personal computer using a different (e.g., Intel) processor, emulator 1303 fetches one or a sequence of binary-image program instructions from storage medium 62 and converts these program instructions to one or more equivalent Intel binary-image program instructions. The emulator 1303 also fetches and/or generates graphics commands and audio commands intended for processing by the graphics and audio processor 114, and converts these commands into a format or formats that can be processed by hardware and/or software graphics and audio processing resources available on host 1201. As one example, emulator 1303 may convert these commands into commands that can be processed by specific graphics and/or or sound hardware of the host 1201 (e.g., using standard DirectX, OpenGL and/or sound APIs).

An emulator 1303 used to provide some or all of the features of the video game system described above may also be provided with a graphic user interface (GUI) that simplifies or automates the selection of various options and screen modes for games run using the emulator. In one example, such an emulator 1303

may further include enhanced functionality as compared with the host platform for which the software was originally intended.

Figure 20B illustrates an emulation host system 1201 suitable for use with emulator 1303. System 1201 includes a processing unit 1203 and a system memory 1205. A system bus 1207 couples various system components including system memory 1205 to processing unit 1203. System bus 1207 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. System memory 1207 includes read only memory (ROM) 1252 and random access memory (RAM) 1254. A basic input/output system (BIOS) 1256, containing the basic routines that help to transfer information between elements within personal computer system 1201, such as during start-up, is stored in the ROM 1252. System 1201 further includes various drives and associated computer-readable media. A hard disk drive 1209 reads from and writes to a (typically fixed) magnetic hard disk 1211. An additional (possible optional) magnetic disk drive 1213 reads from and writes to a removable "floppy" or other magnetic disk 1215. An optical disk drive 1217 reads from and, in some configurations, writes to a removable optical disk 1219 such as a CD ROM or other optical media. Hard disk drive 1209 and optical disk drive 1217 are connected to system bus 1207 by a hard disk drive interface 1221 and an optical drive interface 1225, respectively. The drives and their associated computer-readable media provide nonvolatile storage of computer-readable instructions, data structures, program modules, game programs and other data for personal computer system 1201. In other configurations, other types of computer-readable media that can store data that is accessible by a computer (e.g., magnetic cassettes, flash memory cards, digital video disks,

0960-4760

Bernoulli cartridges, random access memories (RAMs), read only memories (ROMs) and the like) may also be used.

A number of program modules including emulator 1303 may be stored on the hard disk 1211, removable magnetic disk 1215, optical disk 1219 and/or the ROM 1252 and/or the RAM 1254 of system memory 1205. Such program modules may include an operating system providing graphics and sound APIs, one or more application programs, other program modules, program data and game data. A user may enter commands and information into personal computer system 1201 through input devices such as a keyboard 1227, pointing device 1229, microphones, joysticks, game controllers, satellite dishes, scanners, or the like. These and other input devices can be connected to processing unit 1203 through a serial port interface 1231 that is coupled to system bus 1207, but may be connected by other interfaces, such as a parallel port, game port Fire wire bus or a universal serial bus (USB). A monitor 1233 or other type of display device is also connected to system bus 1207 via an interface, such as a video adapter 1235.

System 1201 may also include a modem 1154 or other network interface means for establishing communications over a network 1152 such as the Internet. Modem 1154, which may be internal or external, is connected to system bus 123 via serial port interface 1231. A network interface 1156 may also be provided for allowing system 1201 to communicate with a remote computing device 1150 (e.g., another system 1201) via a local area network 1158 (or such communication may be via wide area network 1152 or other communications path such as dial-up or other communications means). System 1201 will typically include other peripheral output devices, such as printers and other standard peripheral devices.

In one example, video adapter 1235 may include a 3D graphics pipeline chip set providing fast 3D graphics rendering in response to 3D graphics commands

issued based on a standard 3D graphics application programmer interface such as Microsoft's DirectX 7.0 or other version. A set of stereo loudspeakers 1237 is also connected to system bus 1207 via a sound generating interface such as a conventional "sound card" providing hardware and embedded software support for generating high quality stereophonic sound based on sound commands provided by bus 1207. These hardware capabilities allow system 1201 to provide sufficient graphics and sound speed performance to play software stored in storage medium 62.

All documents referenced above are hereby incorporated by reference.

While the invention has been described in connection with what is presently considered to be the most practical and preferred embodiment, it is to be understood that the invention is not to be limited to the disclosed embodiment, but on the contrary, is intended to cover various modifications and equivalent arrangements included within the scope of the appended claims.

0068219.0003